III. 2D ГЕОМЕТРИЧЕСКИЕ ПРЕОБРАЗОВАНИЯ

Двумерные геометрические преобразования – это набор математических операций, используемых для изменения положения, размера и ориентации фигур в двумерной плоскости. Эти операции необходимы в компьютерной графике, поскольку они позволяют манипулировать графическими объектами для создания анимации, применения визуальных эффектов и позиционирования элементов в сцене. К основным геометрическим 2D-преобразованиям относятся ***перенос, вращение, масштабирование, сдвиг*** и ***отражение***.

# 3.1 Перенос

Перенос – это процесс перемещения объекта без изменения его формы, размера или ориентации. Это делается путем добавления константных значений к координатам каждой точки объекта.

Перенос в p5.js осуществляется с помощью функции **translate().**

**Функция Translate():** Перемещает систему координат. По умолчанию начало координат системы координат находится в точке с координатами (0, 0) (верхний левый угол) холста в 2D-режиме и в центре холста в режиме WebGL. Все, что нарисовано после вызова translate(), будет отображаться как перемещённое. Существует два способа вызова translate(), в зависимости от его параметров.

Первый способ вызова **translate()** использует числа для выполнения переноса по положительным осям x и y. Третий параметр, z, является необязательным и используется в 3D-графике.

Второй способ вызова translate() использует объект p5. Vector для выполнения переноса по осям в зависимости от заданного вектора.

По умолчанию преобразования накапливаются. Например, двойной набор **translate(10, 0)** дает тот же эффект, что и однократный набор **translate(20, 0).**

**Примечание.** Преобразования сбрасываются в начале цикла рисования.

**Синтаксис:**

**translate(x, y, [z]);**

**translate(vector);**

где:

x – целое число, обозначающее смещение по положительной оси x;

y – целое число, обозначающее сдвиг по положительной оси y;

z – целое число, обозначающее сдвиг по положительной оси z;

vector – р5. Вектор, на основе которого осуществляется смещение.

Примеры использования функции **translate()** приведены в таблице 3.1.

Таблица 3.1. Примеры использования функции  **translate()**

|  |  |
| --- | --- |
| ***Программа*** | ***Результат*** |
| **translate**(30, 20);  **rect**(0, 0, 55, 55);  //эквивалент rect(30,20,55,55); |  |
| **rect**(0, 0, 55, 55); //Рисует прямоугольник, с координатой верхнего левого угла 0, 0  **translate**(30, 20);  **rect**(0, 0, 55, 55); //Рисует прямоугольник, с координатой верхнего левого угла 30, 20  **translate**(14, 14);  **rect**(0, 0, 55, 55); //Рисует прямоугольник, с координатой верхнего левого угла 44 = 30 + 14; 34 = 20 + 14 |  |
| function **draw**() {  **background**(200);  **rectMode**(CENTER);  **translate**(width / 2, height / 2);  **translate**(p5.Vector.**fromAngle**(**millis**() / 1000, 40));  **rect**(0, 0, 20, 20);  } |  |

Ниже приведен пример выполнения переноса.

**Пример:**

let x = 0;

let у = 0;

let dim = 80.0;

function setup() {

createCanvas(500, 300);

noStroke();

}

function **draw()** {

background(240);

// Анимирует рост значения x

x = x + 0.8;

// Если фигура выходит за пределы холста, сбросьте положение

if (x > width + dim)

{

x = -dim;

}

// Несмотря на то, что наша команда "rect" рисует фигуру с центром в начале координат, перенос перемещает ее в новое положение x и y

translate(x, height / 2 - dim / 2);

fill(255);

**rect**(-dim / 2, -dim / 2, dim, dim);

// Преобразования накапливаются. Обратите внимание, что этот «прямоугольник - rect» движется в два раза быстрее другово, но имеет тот же параметр для значения оси x

translate(x, dim);

fill(0);

rect(-dim / 2, -dim / 2, dim, dim);

}

Результат выполнения кода показан на рисунке 3.1.

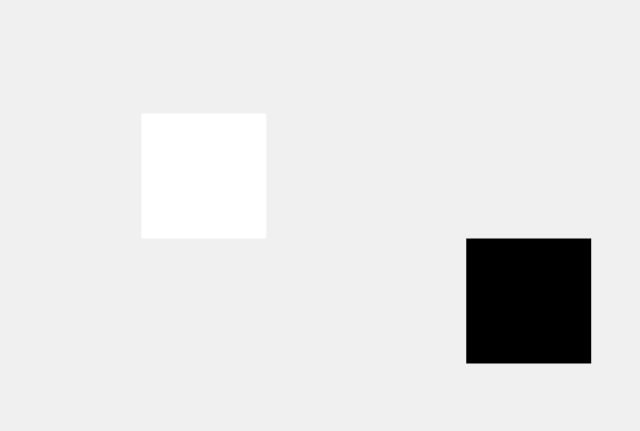


Рисунок 3.1. Пример использования функции **translate()**

# 3.2. Вращение

Вращение включает в себя вращение объекта вокруг фиксированной точки (обычно начала координат) под определенным углом. При этом изменяется ориентация объекта без изменения его размера.

**Функция rotate():** Выполняет геометрическое преобразование вращения. Функция **rotate()** изменяет ориентацию, вращая систему координат вокруг начала координат. Все, что нарисовано после вызова **rotate(),** будет выглядеть повернутым.

Первый параметр указывает угол, на который поворачивается фигура, следующая за вызовом функции **rotate(),** которая интерпретирует значения углов либо градусов, либо радианов в зависимости от текущих настроек функции **angleMode()**.

Объекты всегда вращаются вокруг своего относительного положения относительно начала координат, положительные значения поворачивают объекты по часовой стрелке. Преобразования применяются к функциям, которые следуют за ними, а последующие вызовы функции накапливают эффект. Например, вызов rotate(HALF\_PI), а затем rotate(HALF\_PI) – это то же самое, что и rotate(PI). Все преобразования сбрасываются при повторном запуске draw().

Технически, **rotate()** умножает текущую матрицу преобразования на матрицу вращения.

**Синтаксис:**

**rotate(angle, [axis]);**

где:

angle – угол поворота, заданный в радианах или градусах, в зависимости от текущего угла, заданного angleMode;

axis – (в 3D) задает ось, вокруг которой будет вращаться фигура, это необязательно.

**Примечание.** Преобразования сбрасываются в начале цикла рисования.

Примеры использования и результатов выполнения кода функции rotate**()** приведены в таблице 3.2

Функции **rotateХ(), rotateY(), rotateZ():** вращаются вокруг осей X, Y и Z соответственно.

**Синтаксис:**

**rotateX(angle);**

**rotateY(angle);**

**rotateZ(angle);**

где:

angle – угол поворота, указанный в радианах или градусах, в зависимости от режима, установленного в текущем режиме.

Примеры использования функций приведены в таблице 3.2.

Таблица 3.2. Примеры использования функций **rotate()**

| ***Программа*** | ***Результат*** |
| --- | --- |
| **translate**(width / 2, height / 2);  **rotate(**PI **/ 3.0);**  **rect**(-26, -26, 52, 52); |  |
| function **draw**() {  **background**(255);  **rotateX(millis() / 1000);**  **box**(); |  |
| function **draw**() {  **background**(255);  **rotateY(millis() / 1000);**  **box**(); |  |
| function **draw**() {  **background**(255);  **rotateZ(millis() / 1000);**  **box**(); |  |

В приведенном ниже примере прямоугольник динамически поворачивается вокруг своего центра. Преобразование осуществляется хаотично каждую четную секунду, создавая впечатление вибрации или легкого движения:

let angle = 0.0;

let jitter = 0.0;

function **setup**() {

**createCanvas**(300, 300);

**noStroke**();

**fill**(255);

// Рисуется прямоугольник в центра холста который вращается

**rectMode**(CENTER); }

function **draw**() {

**background**(220);

// Каждую четную секунду переменная «джиттера – вибрации» обновляется на случайное значение в диапазоне от -0,1 до 0,1. Это приводит к изменению угла поворота

if (**second**() % 2 === 0) {

jitter = **random**(-0.1, 0.1);    }

angle = angle + jitter;

// Косинус текущего угла рассчитывается для более плавного движения путем чередования небольших вращений по часовой стрелке и против часовой стрелки, когда вибрация не применяется

let c = cos(angle);

// Фигура перемещается к центру холста

**translate**(width / 2, height / 2);

**rotate**(c);

**rect**(0, 0, 180, 180);  }

Результат работы программы представлен на рисунке 3.2.

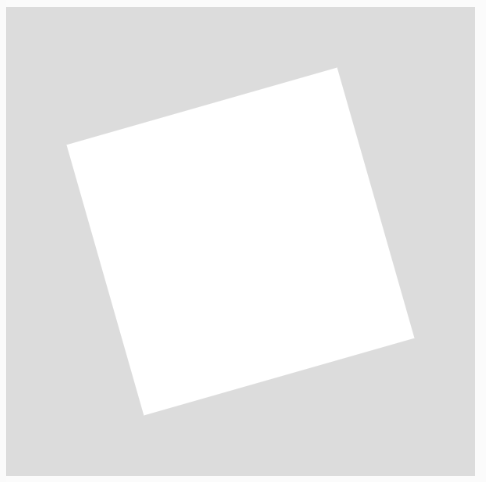


Рисунок 3.2. Результат запуска программы

# 3.3. Масштабирование

Масштабирование – геометрическое преобразование, которое изменяет размер объекта путем умножения координат каждой точки на коэффициент масштабирования. Он может быть равномерным (когда коэффициенты масштабирования по осям X и Yравны) или неравномерными (когда коэффициенты масштабирования разные).

**Функция scale()** – преобразование, которое увеличивает или уменьшает размер фигуры за счет расширения и сужения вершин. Объекты всегда масштабируются относительно их начала координат или системы координат. Коэффициенты масштабирования указываются в десятичных процентах. Например, вызов функции **scale(2.0)** увеличивает размер рисунка на 200%.

Преобразования применяются ко всем фигурам, вызываемым функцией **scale**. Функция имеет кумулятивный эффекта, например, вызов scale(2.0), затем scale(1.0) то же самое, что и scale(3.0), если scale() вызывается в draw(), преобразование сбрасывается при повторном запуске цикла.

Использование этой функции с параметром **Z** доступно только в режиме WEBGL.

**Синтаксис:**

**scale(s, [y], [z]);**

**scale(scales);**

где:

s – процент масштабирования объекта или процент масштабирования объекта по оси X, если задано более одного аргумента;

y – процент масштабирования объекта по оси **Y** (опционально);

z – процент масштабирования объекта по оси**Z** (только для WebGL) (опционально);

scales – процент по осям для масштабирования объекта.

Пример использования функции scale приведен в таблице 3.3. Если в функции scale мы укажем только один параметр, то будет выполнено равномерное масштабирование (***sx = sy***), если укажем два разных параметра, то масштабирование будет неравномерным.

Таблица 3.3. Использование функции **scale()**

|  |  |
| --- | --- |
| ***Программа*** | ***Результат*** |
| **rect**(30, 20, 50, 50);  **scale(0.5);** // равномерное масштабирование  **rect**(30, 20, 50, 50); |  |
| **rect**(30, 20, 50, 50);  **scale(0.5, 1.3);** // неравномерное масштабирование  **rect**(30, 20, 50, 50); |  |

В p5.js существуют специальные механизмы, такие как: глобальная переменная **frameCount** и **millis(),** которые используются для измерения времени или для управления последовательностями анимации и временными событиями в сценах. Несмотря на то, что оба измеряют время, у каждого из них есть свое применение:

**• frameCount** – глобальная системная переменная, которая подсчитывает, сколько кадров (фреймов) было отрисовано с начала выполнения сцены (программы). Она указывает на количество выполнений функции draw() с момента инициализации программы. FrameCount начинается со значения 0 в setup() и увеличивается с каждым полным выполнением функции draw(), это полезно при создании анимаций или временных эффектов, поскольку каждое значение представляет определенный момент времени (с начала программы);

**• millis()** – функция, возвращающая количество миллисекунд, прошедших с момента начала работы программы. Функция millis() отслеживает, как долго был выполнен скетч, в миллисекундах (тысячных долях секунды). Эта информация часто полезна для синхронизации событий и анимаций.

Функция millis() начинает отслеживать время перед запуском кода в setup(). Если код рисунка включает функцию preload(), millis() начинает отслеживать время, как только код в preload() начинает выполняться. Она используется для измерения реального времени и более точна для времени или для событий, не зависящих от количества кадров, на неё не влияют колебания frameCount, это точное измерение прошедшего времени. Она используется для создания анимаций на основе времени или триггерных событий через определенный промежуток времени.

**3.4. Другие виды преобразований**

Помимо основных геометрических преобразований, в p5.js существуют и другие преобразования, такие как **сдвиг**, **зеркальное отражение** и **составление** преобразований.

**Функция shearX():** Сдвиг или растяжение деформирует фигуру вокруг оси X на значение, указанное параметром angle. Углы должны быть указаны в текущем angleMode(). Объекты всегда деформируются относительно их точки привязки, и вращением по часовой стрелке.

Преобразования применяются ко всем функциям, вызываемыми после, и каждый вызов функции накапливает эффект. Например, вызов shearX(PI/2), а затем shearX(PI/2) – это то же самое, что и shearX(PI). Если shearX() вызывается в draw(), преобразование сбрасывается при повторном запуске цикла.

Технически, shearX() умножает текущую матрицу преобразования на матрицу вращения.

**Синтаксис:**

**shearX(angle);**

где:

angle – угол смещения, указанный в радианах или градусах, в зависимости от текущего угла в angleMod.

**Функция shearY():** То же, что и функция shearX(), только по оси Y.

**Синтаксис:**

**shearY(angle);**

где:

angle – угол смещения, указанный в радианах или градусах, в зависимости от текущего угла в angleMod.

Пример использования и результат выполнения кода функции приведен в таблице 3.4.

Таблица 3.4. Использование функции **shear**()

|  |  |
| --- | --- |
| ***Программа*** | ***Результат*** |
| function **draw**() {  **background**(240);  **translate**(width/4, height/4);  **shearX**(PI/4.0)  **rect**(0, 0, 50, 50);  } |  |
| function **draw**() {  **background**(240);  **translate**(width/4, height/4);  **shearY**(PI/4.0)  **rect**(0, 0, 50, 50);  } |  |

В p5.js функции **push()** и **pop()** используются для сохранения или восстановления настроек стиля и графических преобразований в сцене. Они позволяют изолировать преобразования и стили, примененные к определенным объектам, чтобы они не влияли на остальную часть сцены, и необходимы для создания сложных сцен без нежелательных преобразований на всей сцене.

**Функция push():** сохраняет текущее состояние стиля (цвета, толщина линий и т.д.) и преобразования (переносы, повороты, масштабы) во временном "стеке". Эту функцию можно использовать в начале блока кода, в котором должны быть применены преобразования или стили, специфичные для объекта.

**Функция pop():** Восстанавливает состояние, ранее сохраненное с помощью **push(),** удаляя все изменения стиля или преобразования, примененные после push(). Он используется в конце блока кода для возврата к исходным настройкам, чтобы преобразования, примененные к объекту, не влияли на другие элементы сцены.

Эти функции используются, если в сложной сцене есть несколько объектов, push() и pop(), которые помогают сохранить контроль над преобразованиями. Эти функции позволяют изменять только определенные объекты, не затрагивая глобально координаты, цвета или стиль всего эскиза.

### Лабораторная работа No 3

### Тема: СОЗДАНИЕ ДИНАМИЧЕСКОЙ 2D-СЦЕНЫ

**Задачи работы:**

1. Ознакомление, реализация и использование основных преобразований 2D графики, с использованием функций rotate(), translate() и scale().

2. Создайте динамическую сцену с анимированными объектами, применяя различные графические преобразования для изменения положения, размера и ориентации элементов внутри графической сцены.

3. Эксперименты с анимацией преобразований для создания визуальных эффектов и динамических взаимодействий в 2D-сцене.

4. Использование библиотеки p5.js для реализации необходимого функционала в JavaScript и изучение потенциала этой библиотеки в компьютерной графике.

**Количество часов, необходимое для выполнения – 4 академических часа.**

**Цель работы:** развить практические навыки реализации и использования 2D графических преобразований в динамической сцене, с использованием библиотеки p5.js. Получите практические знания о том, как манипулировать объектами и анимировать их, используя функции **rotate(), translate() и scale()** в p5.js.

**Рабочая задача:** Используя преобразования **translate(),** **rotate()** и **scale(),** выполните задачу согласно варианту.

**Вариант 1. Анимированный маятник:** создайте маятник, который движется слева направо, имитируя естественное движение реального маятника с помощью функции **rotate().** Маятник должен вращаться со временем (**millis()** или **frameCount**).

**• Задача:** Используйте вращение для анимации движения маятника.

**• Совет:** используйте **sin()** для медленных, естественных движений.

**Вариант 2. Солнечная система:** постройте простую модель Солнечной системы, в которой планеты вращаются вокруг Солнца, используя **rotate()** и **translate()**. Каждая планета должна иметь свою собственную орбиту и движение.

**• Задача:** применение вращений и перемещений для моделирования планетарных орбит.

**• Совет:** планеты должны двигаться с разной скоростью.

**Вариант 3. Воздушный змей, управляемый мышью:** создайте анимацию, в которой воздушный змей движется по плоскости экрана, а его веревка автоматически подстраивается в соответствии с положением мыши. Используйте **translate()** и **rotate()** для правильного направления воздушного змея в соответствии с движением мыши.

**• Задача:** использовать **mouseX** и **mouseY** для управления кайтом.

**• Совет:** добавьте слегка плавные движения с помощью **lerp()** для хвоста.

**Вариант 4. Анимируйте растущий цветок:** имитируйте рост цветка на экране. Начните с простого стебля, постепенно добавляя лепестки, которые растут и слегка вращаются. Используйте функции **scale()** и **rotate()** для анимации лепестков.

**• Задача:** анимация роста с помощью графических преобразований.

**• Совет:** контролируйте скорость роста с помощью **frameCount**.

**Вариант 5. Интерактивная мозаика:** Создайте мозаику из прямоугольников, которые меняют цвет и форму при наведении на них мыши. Используйте **translate(),** **rotate()** и **scale()** для изменения внешнего вида каждого прямоугольника при взаимодействии с мышью.

**• Задача:** реализация взаимодействия на основе положения мыши.

**• Совет:** каждый прямоугольник должен независимо реагировать на движение мыши.

**Вариант 6. Оптическая иллюзия с вращением:** для создания графического дизайна, создающего оптическую иллюзию, с помощью кругов или линий, которые постоянно вращаются и масштабируются. Анимация должна быть плавной и создавать впечатление кругового движения.

**• Задача:** создание оптической иллюзии за счет непрерывных вращений.

**• Совет:** используйте **rotate()** и **scale()** для создания интересных эффектов.

**Вариант 7. Фракталы с преобразованиями:** Создайте фрактальную модель, в которой преобразования **rotate()**, **scale()** и **translate()** используются для создания фрактального дерева или повторяющейся геометрической структуры. Структура должна быть сгенерирована рекурсивно.

**• Задача:** использовать рекурсию для генерации фрактала.

**• Совет:** каждый уровень рекурсии должен уменьшать размер фигур.

**Вариант 8. Каскад шаров: Смоделируйте** каскад шаров, падающих на экран и отскакивающих от стены. Каждый шар должен вращаться при падении и немного менять размер при ударе о стену (эффект отскока).

**• Задача:** моделирование силы тяжести и отскока с помощью вращения и масштабирования.

**• Совет:** используйте массив для управления несколькими шарами одновременно.

**Вариант 9. Создание интерактивной анимации Разбивание стекла:** создание интерактивной сцены разбивание стекла, в которой пользователь может управлять преобразованиями объекта (масштабировать, вращать, переводить) с помощью мыши и клавиатуры.

**• Задача:** используйте **mouseX**, **mouseY** для управления движением (перемещение), используйте клавиши для изменения размера (масштабирования) и угла поворота.

**• Совет:** объекты должны реагировать на действия пользователя в режиме реального времени.

**Вариант 10. Динамическая генерация геометрической модели:** создайте сцену, в которой несколько геометрических фигур (кругов, прямоугольников, многоугольников) генерируются и анимируются на основе набора преобразований. Каждый вновь добавленный объект необходимо масштабировать и поворачивать в соответствии с его положением, а также динамически перемещать по экрану.

**• Задача:** фигуры должны быть сгенерированы случайным образом или на основе математической закономерности (например, спираль, сетка).

**• Совет:** Используйте **random()** для генерации позиций или размеров и **frameCount** для анимации объектов на экране.

**Вариант 11. Анимированные аналоговые часы:** Создайте аналоговые часы, которые работают в режиме реального времени с использованием преобразований **rotate()** и **translate().** Стрелки часов должны быть анимированными и указывать точное время, минуты и секунды.

**• Задача:** Используйте **hour()**, **minute(),** **second()** в p5.js для определения положения кеглей.

**• Совет:** Используйте **rotate()** для анимации игл на основе времени.

**Вариант 12. Симуляция гравитации:** создайте симуляцию, в которой несколько объектов падают на экран под действием силы тяжести, а при касании дна подпрыгивают вверх (эффект отскока).

**• Задача:** Использование графических преобразований для масштабирования объектов по мере их приближения к дну (как если бы они были сжаты).

**• Совет:** Объекты следует перемещать вниз в соответствии с гравитацией. Касаясь земли, их необходимо поворачивать или масштабировать, чтобы имитировать эффект отскока.

**Вариант 13. Генеративное художественное творчество:** создайте генеративную художественную композицию, которая использует множественные преобразования (перемещение, вращение, масштабирование) для создания интересного геометрического узора на основе набора простых правил.

**• Задача:** Использование рекурсивной функции или цикла для генерации сложных структур.

**• Совет:** результат должен быть случайным при каждом запуске.

**Пример:**

Создайте анимированную сцену, которая включает в себя как минимум три различных объекта (геометрических фигур), каждый из которых управляется с помощью преобразований translate(), rotate() и scale().

**•** Все три объекта должны быть анимированы (перемещение, вращение и масштабирование должны выполняться непрерывно).

**•** Анимация должна быть плавной и использовать frameCount или millis() для управления движениями:

function **setup**() {

**createCanvas**(600, 600);

**noStroke**();  }

function **draw**() {

**background**(200);

// Объект 1: Вращение и масштабирование прямоугольника

**push**();

**translate**(width / 4, height / 2);

**rotate**(**radians**(frameCount % 360));

**scale**(**sin**(frameCount \* 0.01) + 1.5);

// Динамическое масштабирование

**fill**(255, 0, 0);

**rect**(-50, -50, 100, 100);

**pop**();

// Объект 2: Круг, который движется по оси X и вращается

**push**();

**translate**((frameCount % width), height / 4);

**rotate**(**radians**(frameCount % 360));

**fill**(0, 255, 0);

**ellipse**(0, 0, 80, 80);

**pop**();

// Объект 3: Треугольник – перемещается по диагонали и масштабируется

**push**();

**translate**((frameCount % width) / 2, (frameCount % height) / 2);

**scale**(abs(sin(frameCount \* 0.01)) + 0.5); // Динамическое масштабирование

**fill**(0, 0, 255);

**triangle**(-30, 30, 30, 30, 0, -40);

**pop**(); }

Результат выполнения программы показан на рисунке 3.3.

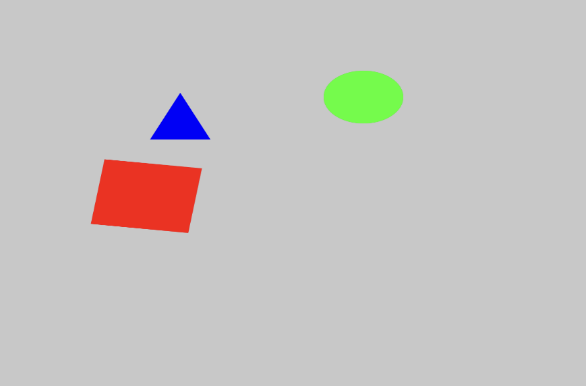


Рисунок 3.3. Динамическая 2D-сцена

**Критерии оценbdfybz**

**1. Реализация функций графического преобразования** (30%) – правильное использование функций rotate(), translate() и scale() внутри 2D сцены для создания движущихся и динамических эффектов объектов. Реализуйте творческие преобразования и соответствующие контексту сцены.

**2. Плавность и непрерывность анимации** (10%) – использование frameCount или millis() для управления движением.

**3. Сложность и разнообразие объектов в сцене** (10%) – способ их визуального взаимодействия.

**4. Оригинальность** в создании анимационной сцены и эффективное использование преобразований (10%).

**5. Корректность кода** (20%) – проверка корректности кода, оптимально, без синтаксических и операционных ошибок.

**6. Соблюдение срока сдачи** (10%) – оценка балла по пунктуальности, если работа была сдана в установленный срок.

**7. Оценка знаний** (10%) – пояснения к процессу выполнения работы, которые могут включать описание основных функций и используемой логики.

**Вопросы для самостоятельной проверки знаний**

1. Что такое геометрические преобразования и почему они важны в 2D графике?

2. Как функция translate() влияет на положение объекта в сцене?

3. Какую роль играет функция rotate() в 2D-анимации и вокруг какой точки происходит вращение по умолчанию?

4. Как работает функция scale() и как она влияет на размер и ориентацию объектов.

5. Как функции translate() и rotate() могут быть использованы вместе для вращения объекта вокруг определенной точки сцены?

6. Как можно избежать влияния преобразований на все объекты в сцене?

7. Как использовать функции push() и pop() в p5.js и зачем они нужны при применении графических преобразований?

8. Как можно применить к объекту одновременно несколько графических преобразований?

9. В чем разница между frameCount и millis() и когда было бы полезнее использовать один вместо другого для анимации?

10. Почему важно использовать push() и pop() при применении множественных преобразований к объекту?

11. Как я могу управлять скоростью вращения объекта с помощью frameCount или millis()?

12. Что произойдет, если применить несколько преобразований без использования push() и pop()? Как это влияет на другие объекты в сцене?

13. В чем разница между translate() и rotate() с точки зрения влияния на положение и ориентацию объекта?

14. Как создать циклическую анимацию с помощью функций sin() или cos() для масштабирования или преобразования вращения?

15. Как команды mouseX и mouseY могут быть использованы для добавления интерактивности в сцену?